Apple IIGS
#54: MIDI Drivers

Revised by: Matt Deatherage November 1990
Written by: Jim Mensch May 1989

This Technical Note describes how to write a driver for use with the Apple IIGS MIDI tools.
Changes since May 1989: Noted that MIDI drivers also work with the MIDI Synth tool.

Apple ships two drivers with the MIDI tool set, `APPLE.MIDI` and `CARD6850.MIDI`, respectively. These drivers are adequate for almost all MIDI hardware currently on the market for the Apple IIGS; however, if your hardware is not compatible with either of these drivers, you have to write your own. This Note includes all the information you need to create a MIDI driver. Note that the same drivers that work with MIDI Tools (Tool #32) also work with the MIDI Synth (Tool #35). This Note collectively refers to MIDI Tools and MIDI Synth as the "MIDI tools."

Purpose of the Driver and Description of Hardware Requirements

The Apple MIDI tools communicate to the MIDI world via a simple driver. The driver's function is managing the transmission and reception of single bytes of MIDI data between the tools and the particular MIDI hardware involved. The MIDI tools operate on the assumption that the hardware has a method of interrupting the system when a character has been received and when a character can be transmitted. Since there is quite a bit of overhead in processing MIDI data, and since MIDI data can comes across a standard MIDI bus at a rate of over 3000 bytes per second, it is suggested that you provide a means for your device to buffer a few characters to reduce system overhead caused by interrupts if you are designing hardware to be used with the MIDI tools.

Format of the Driver File

The driver file is a standard OMF load file, which can be created with any of the popular Apple IIGS assemblers. The file must start with a dispatch table that contains the addresses of the standard driver routines. All driver routines must be in the same segment as the dispatch table. The dispatch table should have 13 four-byte entries, each of which contains the address of the appropriate routine minus one. Table 1 contains addresses of routines in the MIDI driver to perform specific functions.

Call	Function
`Init`	Called to initialize the port and prime the driver
`ShutDown`	Called to close the port and clean up after driver
`Reset`	Called at reset time by the MIDI tools
`IntHandler`	Called when your interrupt occurs
`PollRecv`	Poll input the port for data
`RecvIntOn`	Turns on receiver interrupts
`RecvIntOff`	Turns off receiver interrupts
`PollXmit`	Polls the transmitter to see if another character can be sent
`XmitIntOn`	Enables transmitter interrupts
`XmitIntOff`	Disables transmitter interrupts
`NotImp`	Currently unused
`NotImp`	Currently unused
`NotImp`	Currently unused

Table 1–MIDI Driver Function Routines

Routine Calling Conventions

All driver routines are called with full 16-bit mode enabled and should exit the same way. On entry to each routine, the accumulator contains the direct page pointer that the driver should use if it wants to use the MIDI Tools' or MIDI Synth's direct page. It is the driver's responsibility to set the direct page register and restore it on exit. All other parameters are passed on the stack and should be removed from the stack before the routine exits. The MIDI tools set aside 128 bytes of space on the passed direct page for use by the driver. They are bytes $80–$FF.

If you want to report an error inside of any routine (except `IntHandler`), set the carry flag on exit and load the accumulator with the error code. Use predefined error codes whenever possible. If you need to report a device specific error, use errors in the range $C0–$FF. The MIDI tools will set the high byte of the error code properly for you, so you do not need to do it yourself. Table 2 lists all of the potential predefined error codes.

Error Code	Error Definition
`miToolsErr` ($2004)	The required tools were not started
`miNoBufErr` ($2007)	No buffer is currently allocated
`miDevNotAvail` ($2080)	Requested device is not available
`miDevSlotBusy` ($2081)	Requested slot is already in use
`miDevBusy` ($2082)	Requested device is already in use
`miDevOverrun` ($2083)	Device overrun by incoming MIDI data
`miDevNoConnect` ($2084)	No connection to MIDI
`miDevReadErr` ($2085)	Framing error in received MIDI data
`miDevVersion` ($2086)	ROM version is incompatible with driver

`miDevIntHndlr` ($2087)	Conflicting interrupt handler installed

The Driver Routines

Init

This routine is called by the MIDI tools when it wants to initialize your port and tell the driver to prepare itself for the rest of the calls. Figure 1 shows how the stack looks on entry to this call.

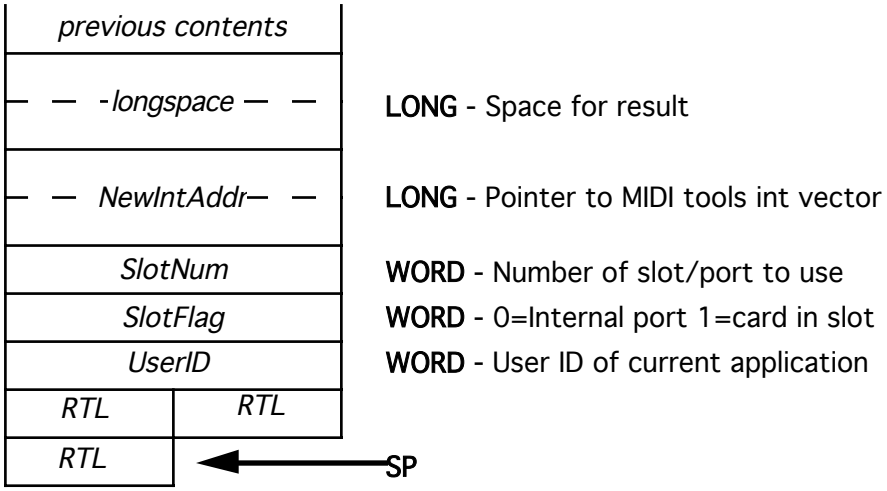

Figure 1–The Stack on Entry to Init

The `Init` routine should first test to see if the port specified by `SlotFlag` and `SlotNum` is available for use. `SlotNum` is the number of the slot or the port that the user has requested for use, and `SlotFlag` indicates whether it is a built-in port or a card in a slot. After determining that the requested device is available, you should initialize the device, allocate any memory that your driver may require (beyond what is available in the direct page), and set the proper system interrupt vector to the address passed in `NewIntAddr`. Before setting the vector, be sure to save the old value, as the MIDI tools expect the result from this routine to be the old address stored in the vector. On exit, the stack should contain the return address and the old vector address.

ShutDown

This routine is called when the MIDI tools want your driver to release the MIDI device and prepare to be unloaded. Figure 2 shows how the stack looks on entry to this call.

Figure 2–The Stack on Entry to ShutDown

Your routine should change the interrupt vector that you used to `OldIntVector`. It should then deallocate all the memory that it allocated, disable all interrupts on the device, and if needed, tell the system that you are no longer using the port in question.

Reset

This routine is called when the system has been reset by the user. Figure 3 shows how the stack looks on entry to this call.

Figure 3–The Stack on Entry to Reset

All you should do at this point is attempt to deallocate any memory you were using and disable interrupts on the device you were using.

Note: Do not set the interrupt vector to `OldIntVector`, instead remove the value from the stack and dispose of it.

IntHandler

The `IntHandler` routine is called by the MIDI tools when an interrupt occurs for the vector that you are using. The MIDI driver performs some setup then calls your routine. This routine does not have any parameters on the stack.

Once called, your `IntHandler` routine should test the port to see if an interrupt has occurred on your device. If your device did not cause the interrupt, you should set the carry and exit as quickly as possible, reducing the system interrupt overhead.

If your device caused the interrupt, you should test the receiver to see if any bytes of data are waiting to be read. If there is data waiting, you should load that data into the accumulator and perform a `JSL` to the following code:

```
InBufGlue    PEA $0400
             PHD
             RTL
```

This code calls the MIDI tools and tell them to accept the character in the accumulator into its input buffer. After accepting the data, control is passed back to the instruction following your `JSL`. If you received a byte of data and an error occurred during reception, you should load the number of the error code into the `y` register and perform a `JSL` to the following code:

```
InErrGlue    PEA $0500
             PHD
             RTL
```

Again, you will regain control right after the `JSL`. Once in your interrupt routine, you may perform the calls above for as much data as you like. For example, if your device has a three-byte buffer, you could call `InBufGlue` once for each waiting character, thus reducing your interrupt overhead and possibly preventing unneeded interrupts.

If the transmitter on your device is ready to send data, you should perform a `JSL` to the following code:

```
OutBufGlue    PEA $8400
              PHD
              RTL
```

This routine will return with the carry set if no data is waiting to be transmitted or the carry clear if data is available. If data is waiting, the next character to send will be in the accumulator, and you should simply send it at that time. If no more data is available, you should disable transmitter interrupts and exit. The MIDI tools will re-enable transmitter interrupts the next time it has data to send.

PollRecv

The `PollRecv` (Poll Receive) routine is called by the MIDI tools every now and then to see if any data might be waiting to be read. There are no parameters on the stack for this call. Your driver should test to see if any data is available and transmit it all to the MIDI tools via the `InBufGlue` described in the `IntHandler` description.

PollXmit

The `PollXmit` (Poll Transmit) routine is called by the MIDI tools when any data is added to the MIDI output buffer. There are no parameters on the stack for this routine. Your driver should enable transmitter interrupts, test to see if it can send any data immediately, and if it can, call `OutBufGlue` as described int the `IntHandler` description to get data to send.

XmitIntOn and RecvIntOn

These routines are called when the MIDI tools want to explicitly enable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, enable transmitter interrupts for `XmitIntOn` and receiver interrupts for `RecvIntOn`.

XmitIntOff and RecvIntOff

These routine are called when the MIDI tools want to explicitly disable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, disable transmitter interrupts for `XmitIntOff` and receiver interrupts for `RecvIntOff`.

NotImp

These routines are not yet implemented, but your driver should be ready to handle a call to them. When called, they should clear the accumulator, clear the carry and perform an `RTL` back to the MIDI tools.

A MIDI Driver Skeleton

You can use the following sample code as a basis for a MIDI driver. It is not a complete driver in itself, and you will need to add code where comments with asterisks (***) appear for it to be functional. This example is in MPW IIGS assembler format.

```
*****************************************************************************
* MIDI.DRVR.Aii
*
* (C)  Copyright Apple Computer, Inc. 1988
* All rights reserved.
*
* by Don Marsh & Jim Mensch
* 10/26/88
*
* This is a shell that can be used to create custom MIDI drivers for use with
* the Apple MIDI tool set. This shell is not functional, but can be used as a
* starting point for creating your own custom MIDI drivers.
*
* Files:    System Macros and equates
*
*
*
* Modification History:
*
* Version 1.0   Mensch
*
*      10/26/88
*
*      Create first draft
*
*****************************************************************************
       Include 'E16.MIDI'
       Include 'M16.MiscTool'
       Include 'E16.MiscTool'
       Include 'M16.util'

;
; Direct page usage Note:
; MIDI drivers may use the upper half ($80-$FF) of the MIDI direct page. When
; a MIDI driver routine is called the Accumulator will contain the direct page
; pointer for the MIDI tool set. If your driver requires more storage than
; 128 bytes, it will have to allocate them itself using the memory manager.

theuserID     equ $80             ; location to store the passed user ID
PortInUse     equ theuserID+2     ; storage for the port number in use
deref         equ PortInUse+2
Temp          equ Deref+4
              EJECT
```

```
*******************************************************************************
*
DispatchTable RECORD
*
* Description:      Every MIDI Driver must start with a driver dispatch table
*      that contains the entry point minus 1 of each of the
*      required entry points.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
              Import DRVRInit
              Import DRVRShutDown
              Import DRVRReset
              Import DRVRIntHandler
              Import DRVRPollRecv
              Import DRVRRecvIntOn
              Import DRVRRecvIntOff
              Import DRVRPollXmit
              Import DRVRXmitIntOn
              Import DRVRXmitIntOff
              Import DRVRNotImplemented
*
* Entry Points:     None
*
*******************************************************************************

              DC.L DRVRInit
              DC.L DRVRShutDown
              DC.L DRVRReset
              DC.L DRVRIntHandler
              DC.L DRVRPollRecv
              DC.L DRVRRecvIntOn
              DC.L DRVRRecvIntOff
              DC.L DRVRPollXmit
              DC.L DRVRXmitIntOn
              DC.L DRVRXmitIntOff
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented


; a few of the routines will need a temporary storage location that can be used
; even after the direct page is set back to what it was, This is a good place
; to put it!

ErrorCode     ds.W 1              ; temporary holder of an error code
              EndR

              EJECT
```

```
*******************************************************************************
*
DRVRInit      PROC
*
* Description:      This is called by the MIDI Tools when it needs to Init
*      your MIDI Driver. This is usually in response to a MIDIxxx
*      call made by the application.
*      When this routine is called, you should allocate any buffer
*      space that you will need beyond the direct page, you should
*      enable the interrupts on your MIDI Device, and then set the
*      appropriate system interrupt vector and return the old vector
*      value. If the init works fine, clear the carry and return.
*      If an error occurs return the appropriate error code
*      in the Accumulator, and set the carry.
*
*
* Inputs:     UserID:Word         ID of application, for mem allocation
*             SlotFlag:Word       0 for internal port/ 1 for slot
*             SlotNum:Word        number of slot/port to use
*             NewIntVector:Long   address to give system as its new
*                                 interrupt vector. This routine is in the
*                                 MIDI tool set, and it performs needed
*                                 setup before it calls your interrupt
*                                 routine
*
* Outputs:    OldIntVector:Long   Address interrupt vector used to have
*
* External Refs:     None
*
* Entry Points:      None
*
*******************************************************************************
; Offsets for parameters on the stack

ProcStatus    equ 1
OldDPage      equ ProcStatus+1
ReturnAddress equ OldDPage+2
UserID        equ ReturnAddress+3
SlotFlag      equ UserID+2
SlotNum       equ SlotFlag+2
NewIntVector  equ SlotNum+2
OldIntVector  equ NewIntVector+4
ParmBytes     equ 10
ParmEnd       equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt vectors
; and enabling interrupt generating hardware. We wouldn't want an interrupt to go
; off before we were ready to handle it! Then set us up to use the MIDI direct
; page.

              php                 ; save the old proc status
              phd                 ; save the old direct page
              tcd                 ; Set Direct page to the one passed
              SEI                 ; and disable interrupts

; now get the user ID and save it, and allocate any buffers that we may need
; Since most drivers will never need more than 128 bytes of storage we will
; not allocate any storage space

              lda UserID,s        ; first save the user ID for later
              sta theUserID       ; in our section of the MIDI DPage

; *** Insert any memory allocation needed here ***
```

```
; Next, you should check the slot flag and number to see if they are compatible
; with this driver. If they are, you should continue and initialize the proper
; port. If they are not proper, you should exit with an error.
; For this example, I will be testing the SlotFlag, to see if it is set to
; external.

             lda SlotFlag,s      ; first test the slot flag to be sure
             bne FlagOK          ; its non-zero.

             ldy #miDevNotAvail  ; if its zero, signal not available
             bra InitError       ; and exit via error routine

FlagOK       lda SlotNum,s       ; Now save the slot number in
             sta PortInUse       ; our data area

; *** At this point you should test the firmware in the desired slot to be sure
; that the card you want is properly installed, if it is not then you should
; pass back the appropriate error ***

; Now that you know that you have the proper slot information and you have tested
; to be sure that you have the hardware needed for the driver it is time for you
; to initialize the interface and to enable its interrupts.

; *** Install code to initialize your hardware/interrupts here ***

; Now that the Port has been properly initialized, you must set up the proper
; system interrupt vector. Since we required an external card above it would
; make sense that you need to use the "Other unspecified interrupt handler"
; vector (Number $0017). But first, remember to get the original vector pointer
; because we must return it to the MIDI tools.

             PushLong #0         ; space for result
             PushWord #otherIntHnd ; vector to retrieve
             _GetVector          ; and get the vector in question
             PullLong Temp       ; place in storage for a sec

             lda Temp            ; now place it on the stack
             sta OldIntVector    ; as the result of this function
             lda Temp+2
             sta OldIntVector+2

             lda NewIntVector    ; now move the MIDI Interrupt routine
             sta Temp            ; pointer into temporary storage
             lda NewIntVector+2
             sta Temp+2

             PushWord #otherIntHnd ; now set the vector to point to
             PushLong Temp       ; the MIDI drivers interrupt routine
             _SetVector

; The driver is now all set up, pull off the passed parms and we are done!
Done         ldy #0              ; set the error code to 0. No error
;
; This is the alternate label for the Done routine that should be called when
; an error has occurred.
InitError
             lda ReturnAddress,s ; Move the return address below the
             sta ParmEnd,s       ; parameters
             lda ReturnAddress+1,s
             sta ParmEnd+1,s

             pld                 ; get the direct page back
             plp                 ; get the processor status back
```

```
             tsc                 ; now adjust the stack pointer
             sec                 ; so that the parameters are gone
             sbc #ParmBytes
             tcs                 ; now the return address is on Top

             tya                 ; put any error into <A>
             cmp #1              ; set the carry if non-zero
             RTL                 ; and return

             EndP

             EJECT
*******************************************************************************
*
DRVRShutDown  PROC
*
* Description:      This routine will be called whenever the MIDI Tools want
*      to cause your driver to let go of the port it was using.
*
*
* Inputs:     OldIntVector:Long   Address to place back into the system
*                                 interrupt vector you were using
*
* Outputs:    Carry clear if successful
*             Carry set if not, error in <A>
*
* External Refs:
       Import DrvrRecvIntOff
       Import DRVRXMitIntOff
*
* Entry Points:
*
*******************************************************************************
             With DispatchTable

ProcStatus    equ 1
OldDPage      equ ProcStatus+1
ReturnAddress equ OldDPage+2
OldIntVector  equ ReturnAddress+3
ParmBytes     equ 4
ParmEnd       equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt vectors
;  We wouldn't want an interrupt to go off before we were ready to handle it!
; Then set us up to use the MIDI direct page.

             php                 ; save the old proc status
             phd                 ; save the old direct page
             tcd                 ; Set Direct page to the one passed
             SEI                 ; and disable interrupts

             lda #0              ; zero out the temp error code
             sta >ErrorCode
; Now First, re-install the old interrupt vector

             lda OldIntVector    ; get the old vector off the stack
             sta Temp            ; and save it in globals for a sec
             lda OldIntVector+2
             sta Temp+2

             PushWord #otherIntHnd ; now set the vector to point to
             PushLong Temp       ; its original routine.
             _SetVector
```

```
; Next, turn off the interface hardware, and tell it to stop generating
; interrupts. We can share some code here and call our DRVRRecvIntOff and
; DRVRXmitIntOff routines. Always remember load the direct page into the
; accumulator.

             tdc                 ; get direct page into <A>
             jsl DRVRXmitIntOff  ; and turn off transmitter interrupts

             tdc
             jsl DRVRRecvIntOff  ; and now receiver interrupts.

; *** Usually turning off interrupts will be all that you would need to do at
; this point, however, if your interface card requires extra shutdown code
; this is where you would place it ***

; *** If you allocated any memory in the DRVRInit call, this is the place to
; get rid of it.

; If an error were to occur in this routine, you should simply store the error
; number in our temporary error code variable like this
;
;            lda #ErrorNumber
;            Sta >ErrorCode


Done
; Now that we are done shutting down the driver, pull off the passed data
; and end.
             pld                 ; first retrieve the old dpage
             plp                 ; and processor status

             Longa Off           ; next move the return address
             SEP #$20            ; we need a short acc for this trick

             pla                 ; pull the 3 byte return address
             ply                 ; into <A> and <Y>

             plx                 ; now remove the remaining bytes
             plx                 ; of passed parameters

             phy                 ; and restore the return address
             pha

             Longa On
             REP #$30            ; and turn back on full 16-bit mode

             lda >ErrorCode      ; retrieve the error code
             cmp #1              ; and set the carry if non-zero
             RTL
             EndP

             EJECT
```

```
*******************************************************************************
*
DRVRReset     PROC
*
* Description:      This routine will be called whenever MIDIReset is called.
*      and that should only happen when an actual reset occurred.
*      It should in most cases perform the exact same functions
*      as MIDI Shutdown.
*
*
* Inputs:     OldIntVector:Long   Original contents of interrupt vector
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

       jmp DRVRShutDown

       EndP

       EJECT
*******************************************************************************
*
DRVRIntHandler      PROC
*
* Description:      This routine is the very core of the MIDI driver. It takes
*      care of passing data back and forth between the MIDI tools
*      and your hardware. It will be called for both input and
*      output.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
             Import DRVRXmitIntOff
*
* Entry Points:
             Export InBufGlue
             Export InErrGlue
             Export OutBufGlue
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage

; The first thing the interrupt routine should do is to test to see if the
; interrupt was actually generated by our port. If it was then we should handle
; it, but if not, we should simply exit this routine with the carry set as
; fast as we can, so that the next interrupt handler will get it in a timely
; manner.

; *** Insert code here to test to see if the original interrupt was yours ***

             beq ServicePort     ; if it was our, handle it

; If the interrupt was not ours, set the carry and leave
             pld                 ; restore the direct page
             sec
             rtl
```

```
ServicePort                     ; the interrupt was ours, continue

; This routine should test the interrupt again, too see if the port is ready
; to transmit or receive, If it is ready to transmit or receive, it should
; then call the ServiceRecv, or ServiceXMit routines

; *** Insert code here to test for receive

             bne ServiceRecv     ; if chars waiting try receive it

; If no more characters are waiting, see if we are ready to transmit any
; characters.

             bne ServiceXMit     ; if can send a character do it

; If both the above tests fail, then exit the interrupt handler for now
             pld                 ; restore the direct page
             clc                 ; clear the carry to indicate serviced
             RTL                 ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***

; Call MIDI tools this way if no error has occurred on receive (<A> contains the
; data read)
RecvOK
             jsl InBufGlue       ; call the MIDI tools
             bra ServicePort     ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
             ldy #miDevReadErr   ; load Y with the error
             jsl InErrGlue       ; call the midi tools
             bra ServicePort

; The routine ServiceXmit will be called when the port is ready to send data.
; it will actually call the MIDI tools and get a character to send.
ServiceXmit

             jsl OutBufGlue      ; call the MIDI tools for the next char
             bcs NoMoreData      ; if the carry set then no data to send

; *** at this point the byte to transmit is in <A>, place your code to output
; it thru the port here ***


; Now that the data has been sent, you can either loop thru ServicePort again,
; or you could simply end and wait for the next interrupt to send another
; character. This sample will simply exit at this point
       bra Done      ; after sending the character end.
```

```
; NoMoreData is called when the MIDI Tools said that they did not have any more
; data to transmit, so we should turn off transmitter interrupts at this point
; in case our device likes to keep interrupting if its empty.
NoMoreData
             phd                 ; push the direct page reg on the stack
             jsl DRVRXmitIntOff  ; enable xmit interrupts
Done
             pld                 ; restore the DPage
             clc                 ; signal the interrupt as handled
             rtl                 ; and get outta here!

; The routine inbufglue should be called when you received a character from your
; port with no error and you want to pass it to the MIDI tools.
InBufGlue    pea $0400           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools

; The routine inErrGlue should be called when you received a character from your
; port and an error has occurred. In this case, it should still be passed to the
; MIDI driver, as it may still be useful
inErrGlue    pea $0500           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools

; The routine OutBufGlue should be called when you are ready to send a char
; out your port. The MIDI tools will will return with the character to send
; in <A>. If the MIDI tools have no more characters to send then OutBufGlue
; will return with the carry set.
OutBufGlue   pea $8400           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools
             EndP

             EJECT
*******************************************************************************
*
DRVRPollRecv  PROC
*
* Description:      This routine is called by the MIDI tools when it wants to
*      pool the port for data instead of waiting for an interrupt.
*      its function is similar to that of the our interrupt handler
*      except that it only does input.
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
             Import InBufGlue
             Import InErrGlue
*
* Entry Points:     None
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage
             php
             SEI
```

```
ServicePort                     ; the interrupt was ours, continue

; This routine should test the port too see if the port has any data for use
; to receive. If it does, it calls the MIDI tools and hands it off. Also note
; this routine will turn off interrupts, since we wouldn't want any stray
; receiver interrupts to spoil our fun and grab the data from us. (This is
; very important for certain types of ports which may signal that the port
; is ready and the generate an interrupt, thus leaving us in a situation where
; our interrupt routines could steal the interrupt right out from under us before
; we fetched it, thus allowing us to possibly double post a character.

; *** Insert code here to test for received data ***

             bne ServiceRecv     ; if chars waiting try receive it


; If no more data is waiting  exit this routine.
             plp
             pld                 ; restore the direct page
             clc                 ; clear the carry no errors possible
             RTL                 ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***

; Call MIDI tools this way if no error has occurred on receive (<A> contains the
; data read)
RecvOK
             jsl InBufGlue       ; call the MIDI tools
             bra ServicePort     ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
             ldy #miDevReadErr   ; load Y with the error
             jsl InErrGlue       ; call the midi tools
             bra ServicePort
             EndP
             EJECT
```

```
*******************************************************************************
*
DRVRPollXMit  PROC
*
* Description:      This routine is called when the MIDI tools wants to start
*                   an output stream. The tool set calls this routine for the
*                   first character of data, and then this routine is
*                   responsible for enabling transmitter interrupts and sending
*                   the character.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:    None
             Import OutBufGlue
             Import DRVRXmitIntOn
*
* Entry Points:     None
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage
             php                 ; disable interrupts as we are now going
             SEI                 ; to turn on xmitter interrupts.

; First see if the port is ready to send any data, if not simply exit

; *** Insert code here to test if output is ready ***

             bcs Done            ; if not, then simply end

; The port is ready to accept a character for output so, call MIDI tools
; to get the next character

             jsl OutBufGlue      ; get the next character
             bcs Done            ; if carry set, no chars to xmit so end


             pha                 ; save the character to send
             phd                 ; push the direct page reg on the stack
             jsl DRVRXmitIntOn   ; enable xmit interrupts
             pla                 ; retrieve the character to send

; *** Insert code here to transmit a character ***
Done
             plp                 ; get the old interrupt status
             pld                 ; get the old direct page
             lda #0              ; no errors are possible
             clc
             rtl

             EndP

             EJECT
```

```
*******************************************************************************
*
DRVRXmitIntOn PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      enable transmitter interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

              php                 ; save proc status/interrupt state
              phd                 ; save the old direct page
              tcd                 ; use the MIDI tools DPage
              SEI                 ; disable interrupts

; *** Insert code here to enable transmitter interrupts on your device

              pld                 ; recover old direct page
              plp                 ; recover old interrupt state
              lda #0              ; and return no-error (none possible)
              clc
              rtl
              EndP

*******************************************************************************
*
DRVRXmitIntOff      PROC
*
* Description:      This routine will be called when the MIDI tools need to
*                   Disable transmitter interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

              php                 ; save proc status/interrupt state
              phd                 ; save the old direct page
              tcd                 ; use the MIDI tools DPage
              SEI                 ; disable interrupts

; *** Insert code here to Disable transmitter interrupts on your device

              pld                 ; recover old direct page
              plp                 ; recover old interrupt state
              lda #0              ; and return no-error (none possible)
              clc
              rtl
              EndP

              EJECT
```

```
*******************************************************************************
*
DRVRRecvIntOn PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      enable receiver interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

             php                 ; save proc status/interrupt state
             phd                 ; save the old direct page
             tcd                 ; use the MIDI tools DPage
             SEI                 ; disable interrupts

; *** Insert code here to enable receiver interrupts on your device

             pld                 ; recover old direct page
             plp                 ; recover old interrupt state
             lda #0              ; and return no-error (none possible)
             clc
             rtl
             EndP


*******************************************************************************
*
DRVRRecvIntOff      PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      Disable receiver interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

             php                 ; save proc status/interrupt state
             phd                 ; save the old direct page
             tcd                 ; use the MIDI tools DPage
             SEI                 ; disable interrupts

; *** Insert code here to Disable receiver interrupts on your device

             pld                 ; recover old direct page
             plp                 ; recover old interrupt state
             lda #0              ; and return no-error (none possible)
             clc
             rtl
             EndP
```

```
*******************************************************************************
*
DRVRNotImplemented  PROC
*
* Description:      Dummy routine, should leave the stack alone and return
*      no error
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************
             lda #0
             clc
             RTL
             EndP

             END
```

Further Reference:

- *Apple IIGS Toolbox Reference Update*